Reliability Analysis of Compressed CNNs

Stefano Ribes, Alirad Malek, Pedro Trancoso, Ioannis Sourdis
Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden
{ribes, aliradm, ppedro, sourdis}@chalmers.se

Abstract

The use of artificial intelligence, Machine Learning and in particular Deep Learning (DL), have recently become a effective and standard de-facto solution for complex problems like image classification, sentiment analysis or natural language processing. In order to address the growing demand of performance of ML applications, research has focused on techniques for compressing the large amount of the parameters required by the Deep Neural Networks (DNN) used in DL. Some of these techniques include parameter pruning, weight-sharing, *i.e.* clustering of the weights, and parameter quantization. However, reducing the amount of parameters can lower the fault tolerance of DNNs, already sensitive to software and hardware faults caused by, among others, high particles strikes, row hammer or gradient descent attacks, *et cetera*. In this work we analyze the sensitivity to faults of widely used DNNs, in particular Convolutional Neural Networks (CNN), that have been compressed with the use of pruning, weight clustering and quantization. Our analysis shows that in DNNs that employ all such compression mechanisms, *i.e.* with their memory footprint reduced up to 86.3×, random single bit faults can result in accuracy drops up to 13.56%.

Keywords

Machine Learning, CNNs, Fault Tolerance, Compression, Caffe

I. Introduction

In recent years, artificial intelligence, machine learning and in particular deep learning have seen a steady growth in popularity thanks to their ground-breaking results. An example of this can be found in Deep Neural Networks (DNN) and in particular Convolutional Neural Networks (CNNs), a special kind of DNN models. CNNs revolutionized the field of computer vision by classifying with high accuracy images from a large set of possible classes. CNNs do so by repeatedly tensor-multiplying an input image with a set of parameters, called *weights*. The resulting tensor can be then further multiplied for the next set of weights. This is process is repeated for a certain amount of steps, or *layers*, each characterized by its own weight parameters. The final result of the data processed throughout the layers is a probability vector that highlights the corresponding class which the image belongs to.

In order to achieve their high accuracy, CNNs go through a process called *supervised training*, which effectively tunes their parameters by aiming at minimizing the error between expected image labels (training images) and the ones produced by the network. Once trained, a CNN can be utilized to classify images never seen during training. The use and execution of trained networks is usually referred as *inference*.

Since CNNs, and DNNs in general, are typically composed of several layers, made of thousands of weights each, running DNNs inference requires a significant amount of memory accesses, eventually making the memory a performance bottleneck. Because of that, accelerating DNNs on general purpose CPUs, GPUs and FPGAs had proved being a challenging task. This drove a lot of research effort into

This work is supported by the European Commission under the Horizon 2020 Program through the ECOSCALE (grant agreement 671632) and SHARCS (grant agreement 644571) projects as well as by the European Research Council (ERC) under the MECCA project (Contract No. 340328).

compressing the memory footprint of DNNs in order to improve their performance at inference time in terms of execution time.

However, compressing DNNs, and in particular CNNs, can make the networks more sensitive to faults and attacks. For instance, a CNN can be utilized for classifying objects in images coming from a camera sensor. If the camera is mounted on a self-driving car, the CNN inference would require fast computation and high accuracy in detecting possible objects (like street signs or obstacles) [1]. One straightforward way for improving its performance is to reduce its parameter space, for example by using parameter quantization, *i.e.* moving from a floating point representation to a fixed point one, commonly at 16 or 8 bit. In this scenario, a fault happening in any of the layer parameters of the CNN can eventually propagate, due to the network structure, to the following layers, thus possibly affecting the classification task (like not being able to identify an incoming obstacle). The problem can become even more severe because of the compression in place: a fault, or attack, in a fixed point parameter, that results in a bit flip, can cause a change in its value of a greater magnitude compared to faults taking place in floating point values. This can ultimately increase the chances of the CNN producing wrong outputs, and in turn to a lower accuracy.

In this work, we investigate if and how single and multi bit flip faults can have more severe consequences on the output of compressed DNNs running inference, compared to uncompressed networks. Our hypothesis is that faults happening in the compressed network parameters may cause CNNs to misclassify inputs at a higher rate compared to their original, uncompressed, counterparts, thus significantly reducing their accuracy score. Based on that, our final aim is to check the conditions for utilizing Odd-ECC from Malek *et al.* [2]. In fact, Odd-ECC can take advantage of the different fault sensitivity of the data to provide efficient and light-weight ECC protection to the different application data regions.

For our analysis, we explore the sensitivity to faults of several DNNs, with a focus on CNNs widely used in research, that have been compressed down to $\times 86$ their original size, *i.e.* the memory footprint of their weights. In order to perform our analysis, we propose a novel framework, named Caffe Macchiato, which we used to compress the networks and then inject single and multi bit faults in specific data regions. To the best of authors' knowledge, this is the first work to analyze and compare the fault tolerance of several CNNs at different levels of compression.

In the remainder of this work, Section II offers an overview of related works on injecting faults or attacks in deep neural networks. Section III gives the necessary background knowledge behind CNNs' architectures. In Section IV we describe the implementation of Caffe Macchiato and the methodology we followed to measure the sensibility of compressed CNNs against transient faults. Finally, in Section V we evaluate the performance of the networks and show the results of our experiments, before concluding in Section VI with a discussion on our findings.

II. Related Work

Many authors have investigated the robustness of deep neural networks against faults. DNNs can be executed and accelerated on a variety of computing devices, such as GPUs, ASICs and FPGAs. When focusing on faults in accelerators, faults can happen in the accelerator datapath, such as in MAC units [3]–[5], or in buffers [3], [6]. Faults happening in buffers can have a higher impact on DNNs performance than in the datapath counterpart, since buffers are usually used to store partial results of an accumulation and so eventual errors can add up, thereby leading to significant drops in the network accuracy.

Other works focus on studying faults and attacks happening in the network parameters stored in main memory [7], [8]. One example of attacks are trojan attacks on CNNs. Trojan attacks attempt to make the networks misclassify images upon receiving a specific trigger image, while maintaining the same functionality in all the other cases [9]–[12]. Other types of attacks include row hammer, laser beam, gradient descend [13] or backdoor [14], [15] attacks.

DNNs are traditionally utilizing single precision floating point representation for their parameters. However, it has been showed that the accuracy of DNNs is not particularly affected when moving to

a fixed point representation, *i.e.* after performing a parameter quantization. When testing and analyzing the sensitivity to faults, Reagen *et al.* [6] propose a fault-injecting framework that accounts for this change of precision, while in Li *et al.* [3] single and multi bit faults are injected in different positions within the quantized network parameters. Both works show that quantized parameters are more sensible to faults compared to floating point values due to their reduced bit width. Along side floating and fixed point values, particular networks called Binarized Neural Networks (BNN) can utilize parameters reduced to binary or ternary values. Khoshavi *et al.* [16] inject cumulative faults in different parts of a BNN that has been implemented as an FPGA accelerator. In their work, they show that 100 single bit faults can cause a drop of 76.7% when injected in the last fully connect layers. However, these works do not account for networks in which multiple compression techniques are applied. In fact, parameter quantization is orthogonal to pruning and weight sharing.

When it comes to fault detection, prevention and correction instead, Li *et al.* [3] selectively apply latch hardening to DNNs accelerators datapath. Another solution from Qin *et al.* [17] is instead correcting the detected faulty parameters by setting them to zero. Along side with that, they also introduce a binary representation for real numbers that is able to limit the effects of faults. Regarding ECC mechanisms targeting DNNs, Guan *et al.* [18] propose to store error check bits in the unused MSB of quantized CNN 8 bit parameters. Their approach is based on a novel training algorithm that regularizes the spatial distribution of large magnitude weights, allowing to exploit unused space for allocating the ECC bits.

Among approaches for protecting DRAM applications that are not tailored to DNNs, Malek *et al.* [2] work, Odd-ECC, dynamically selects and sets the fault tolerance level of different data regions. The ECC bits are in fact stored in separate physical pages, but are physically aligned with the data they protect. This solution allows to access memory efficiently, reducing the energy consumption and significantly improving memory fault tolerance.

Closest to our analysis is the work of Segee *et al.* [19], in which a feed forward neural network is first pruned and then tested for measuring its fault tolerance. Compared to our work, they are not injecting faults by flipping bits, but rather by zeroing out the faulty weights. Moreover, they only focus on one single-input-single-output feed-forward neural network, whereas we analyze a set of more complex networks classifying images. Finally, we not only consider pruning, but also more advanced compression techniques such as weight sharing and quantization.

III. Background

A. Convolutional Neural Networks

Deep neural networks (DNNs) have been extensively applied to many classes of problems like image classification [20], [21], scene labeling [22] or language translation [23]. Figure 1 shows an example of a Convolutional Neural Network (CNNs), a particular type of DNNs. DNNs are usually composed of several layers, which are represented by the various blocks and rectangles in figure. Layers are typically connected in a pipeline fashion, where data flows from a layer to the next one. The data produced and consumed by a layer is usually referred as either Feature Maps (FM) or *activations*. A layer can contain a set of parameters called weights, which are used to process the incoming data. Other layers can process incoming inputs without requiring weights. Layers including weights are showed as boxes in the figure, they are, for instance, convolutional (CONV) and fully connected (FC) layers. Weight-free layers are instead pictured as rectangles: Max Pooling, Rectified Linear Unit (ReLU) and Softmax layers are some examples of this kind of layers.

Weights can be learned, *i.e.* finetuned, through a process called training. Training allows a DNN to tune its weights to solve a specific problem, such as image classification.

CONV layers in particular perform a convolution of an input feature map with a weight tensor W of dimension $C_o \times K_h \times K_w \times C_i$. The tensor is divided in kernel matrixes of height K_h and width K_w, as illustrated in Figure 2. The terms C_i and C_o represent the number of input and output channels of the

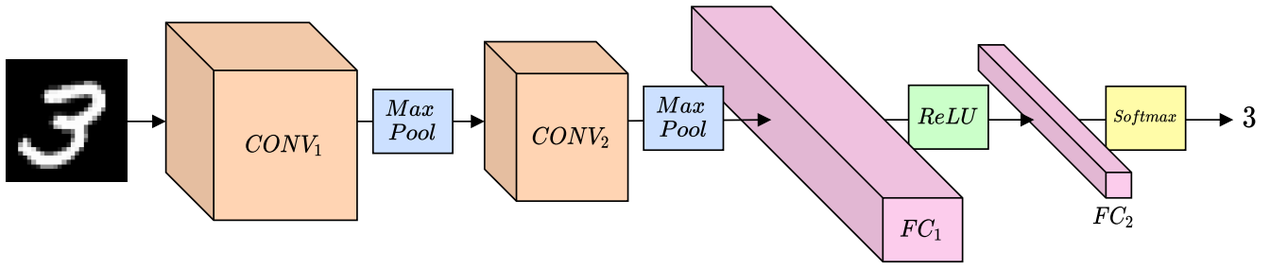

Fig. 1: An example of Convolutional Neural Network (CNN) called LeNet-5, from the work of Le Cun *et al*. The CNN classifies black and white images of hand-written digits [20].

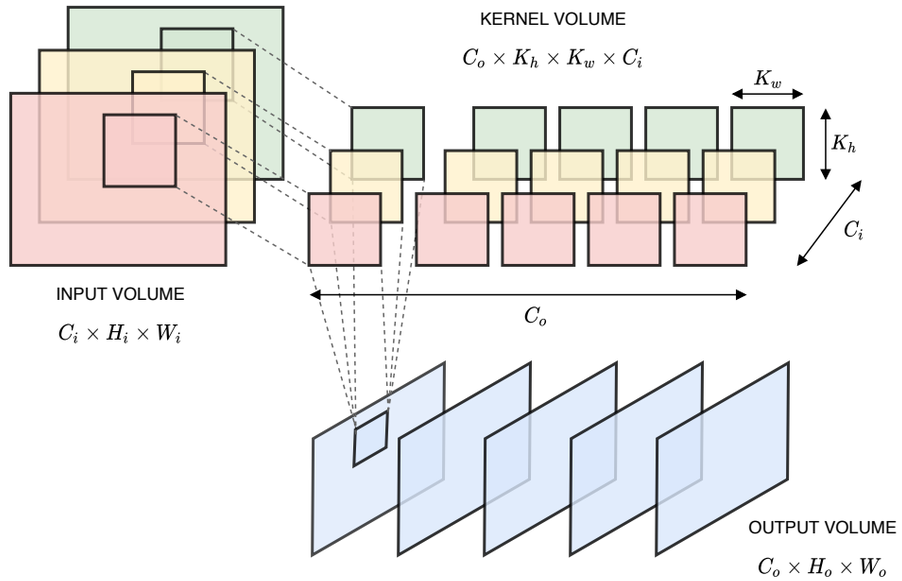

Fig. 2: Input, output and kernel volumes of a CONV layer.

input and output data. Given a Feature Map FM_i as input, *e.g.* an image, a CONV layer produces an output Feature Map FM_o whose elements at coordinates (h_o, w_o, c_o) can be obtained following Equation 1.

$$FM_o(h_o, w_o, c_o) = B(c_o) + \sum_{k_h=0}^{K_h-1} \sum_{k_w=0}^{K_w-1} \sum_{c_i=0}^{C_i-1} FM_i(h_o \cdot S + k_h, w_o \cdot S + k_w, c_i) \cdot W(c_o, k_h, k_w, c_i), \tag{1}$$

where S corresponds to a stride parameter that can further reduce the output feature map dimensions and the amount of operations, while B is a bias term. The idea is to *convolve* the input feature maps with the kernels, *i.e.* weights, saving parameters and preventing overfitting [20]. The usual step that follows the convolution operation is applying an *activation function* to the FM_o. Typically, activation functions are non-linear functions which are applied to the output of a layer before forwarding it to the next one. ReLU is a popular activation function [24] whose behavior is described in Equation 2.

$$\text{ReLU}(x) = max(x, 0) + \gamma \cdot min(x, 0), \tag{2}$$

where γ represents the negative slope of the function.

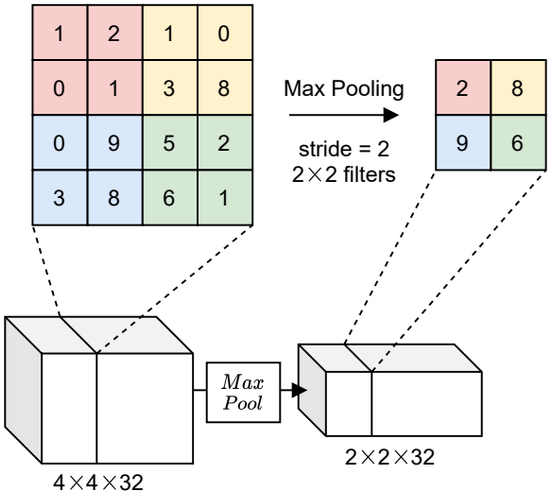

Fig. 3: Visual representation of a Max Pooling layer. Pooling layers downsample a given tensor reducing the number of parameters and improving the network accuracy.

Pooling layers are generally placed after CONV layers and are layers responsible for reducing the size of the feature maps thereby preventing the network from overfitting [25], *i.e.* avoids the network learning only the training data. They do so by downsampling the input tensors, as shown in Figure 3.

After a series of CONV layers, a CNN generally includes a set of FC layers in its ending part, before concluding with a Softmax layer. FC layers perform Equation 3, which consists of a simple matrix-matrix multiplication with a weight matrix W followed by a bias addition B. Like CONV layers output, the FC layers output is further modified by applying an activation function like ReLU.

$$FM_o = FM_i \cdot W + B \tag{3}$$

DNNs including only FC layers are defined as Fully Connected Deep Networks (FCDN). A FCDN typically features two to three FC layers with ReLU activation functions and also terminates with a Softmax layer when solving a classification problem. Finally, a Softmax layer normalizes the output of the last network into a probability distribution over the classes to predict. Hence, the predicted class is identified by picking the class corresponding to the highest probability.

B. DNNs Compression

Modern neural network architectures generally include a large amount of parameters [26]. However, it has been shown that neural networks can tolerate a reduction in the amount of parameters without loosing significant accuracy. Such removal of elements is referred as pruning [27]. There exists in literature several ways of pruning a network [28]. One popular and effective technique for pruning consists of iteratively prune and finetune a network to maintain its accuracy [29].

Once most of the weights are zeroed out and do not contribute anymore to the network execution, they can be further compressed by techniques such as weight-sharing and/or quantization (to fixed point or to half-precision floating point representations). Weight-sharing [29], or network clustering, is a technique that attempts to map a limited number of weights in a layer (referred as *centroids*) to all the rest of the weights. In this way, each layer weight is associated to a specific centroid thanks to an index pointing to it. Because of that, only the centroids and the indexes are required during network execution. The centroids can be generated with a clustering algorithm and then finetuned with a backpropagation algorithm to restore the original network accuracy.

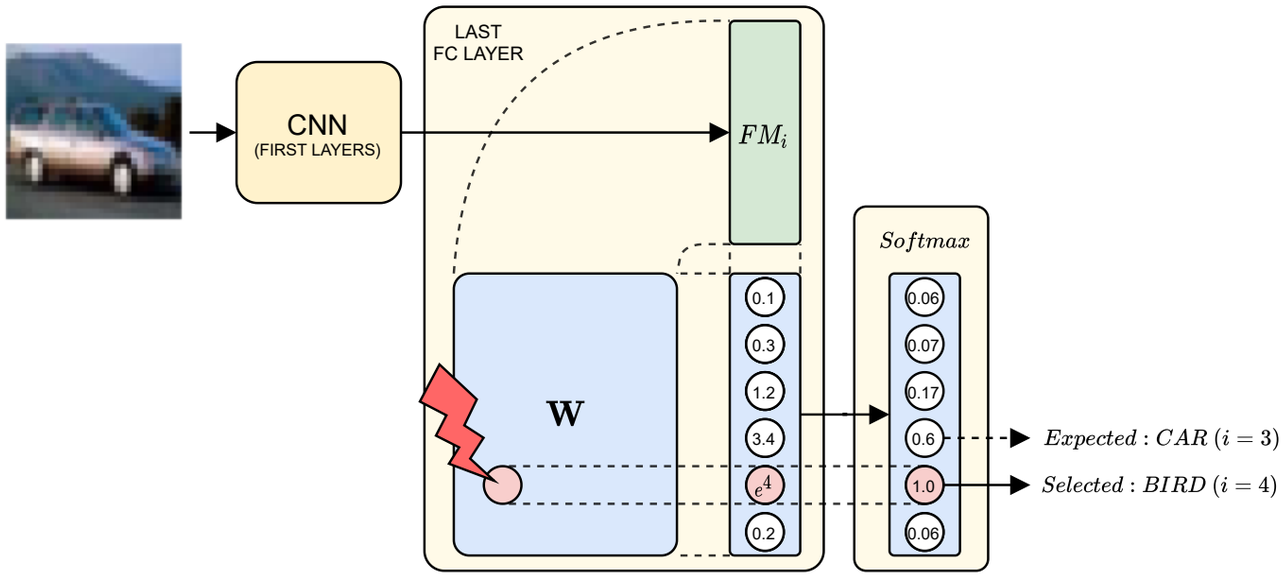

Fig. 4: How a fault in a weight parameter can affect the classification of an image. A fault in any weight parameter can propagate to the final output and cause a misclassification.

Finally, the centroids can be quantized from a floating point 32 or 64 bit representation to 8 or 16 dynamic fixed point representation [30]. The conversion typically causes drops in the network accuracy and therefore requires a retraining phase to calibrate the fixed point parameters.

C. Soft Errors and Accuracy Degradation in DNNs

Focusing on classification tasks, like assigning a label to a given image, the Softmax layer is a popular final layer for DNNs. The Softmax function is responsible for normalizing the values of the output vector into a probability distribution and for highlighting its maximum value. In practice, it applies Equation 4 to all the elements of the output of a DNN.

$$\sigma(x^i) = \frac{e^{x^i}}{\sum_{j=0}^{N-1} e^{x^j}} \ , \ i = 0, 1, ..., N-1 \ , \ x^i \in \mathbb{R}^N \tag{4}$$

where N is the length of the vector and so the number of classes. Once the elements are processed, *i.e.* all normalized in $[0, 1]$ and all adding up to 1, the index of the maximum value is selected to identify the class which the input belongs to. This means that having a maximum value at a different index will cause the network to classify the input into another class.

In presence of a fault in one of the weight parameters, the faulty value might generate, in the DNN's output, a different maximum value than the expect one, thereby altering the network prediction. An example of such scenario is depicted in Figure 4. Since the FC layer performs a vector-matrix multiplication, a fault in any of its weight matrix parameters can propagate to the layer output and so to the final Softmax layer. In case the fault significantly changes the magnitude of one output parameter, the network will select it as being the class with the highest probability. Because of the fault, if the maximum value results in a different class than the expected one, then the input image is misclassified, thus degrading the accuracy of the DNN.

IV. Design and Methodology

In this section we describe the implementation of our proposed framework Caffe Macchiato, its compression scheme, the application data regions that are sensible to faults and finally how we perform the sensitivity analysis of DNNs.

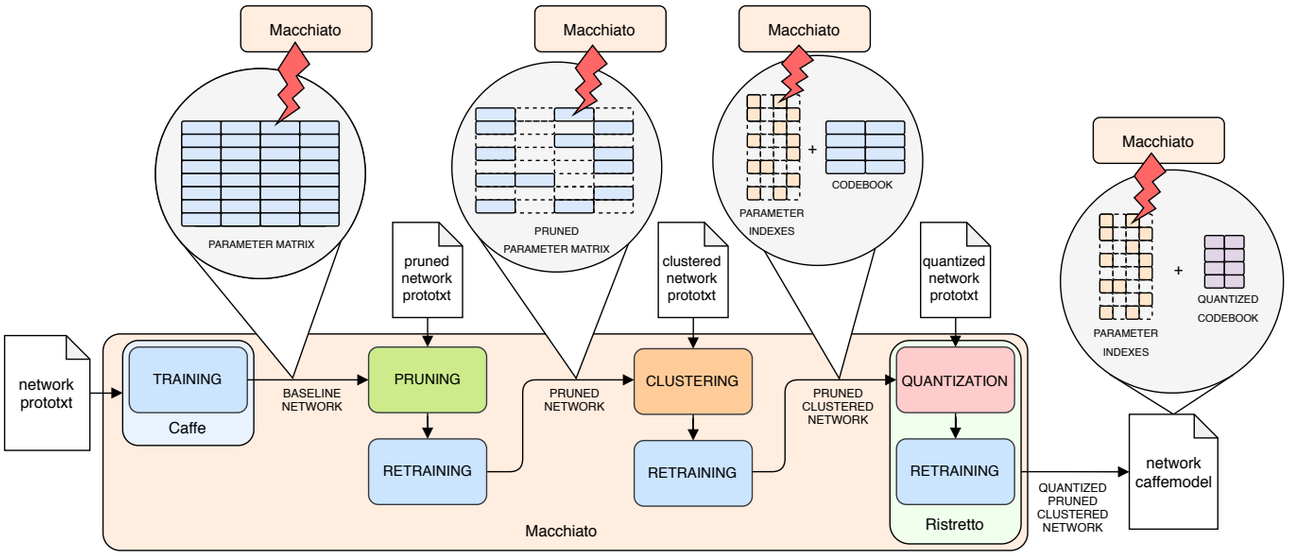

Fig. 5: The Caffe Macchiato framework. After training in Caffe, Caffe Macchiato prunes and clusters the network while Ristretto finally quantizes it. Besides compression, Caffe Macchiato injects faults in any compression step parameters, as indicated by the red lightnings on the top.

A. *Caffe Macchiato Framework*

We modified Caffe [31] and Ristretto [30] to implement the Caffe Macchiato framework, illustrated in Figure 5. In developing Caffe Macchiato, we followed the work of Han *et al.* [29] for designing the steps for network compression without causing significant drops in accuracy. Caffe Macchiato integrates Caffe for training a network given a specification file (in Google Protocol Buffer format [32]), it then performs pruning and clustering before calling Ristretto for the final quantization[1] step. Macchiato also implements a fault injection system for causing bit flips in any compressed network.

Focusing on the implementation details, starting from pruning, we modified the CONV and FC layers in Caffe to include a zero mask to prune the parameters of both weights and bias that are below an adjustable threshold. In order to prune an entire network, Caffe Macchiato first sets the zero masks of all the layers, then retrains the network, *i.e.* finetunes it, while keeping the zero masks fixed.

Once the network is pruned, Caffe Macchiato proceeds applying weight sharing by clustering each layer's weights into a set of k clusters through the K-means algorithm. Caffe Macchiato then stores all the clusters centroids of a layer into a *codebook*, while each weight is substituted by an index, *i.e.* pointer, to its corresponding centroid value. After populating the codebook, Caffe Macchiato retrains the network, thereby finetuning the centroids by following the methodology described in [29], but maintaining the indexes fixed. At run time, the codebook values are used as a substitute of the original weight values, further improving the memory footprint of the network.

Finally, after pruning and clustering, Caffe Macchiato utilizes the Ristretto framework [30] to quantize the network parameters from single precision floating point to a dynamic fixed point representation. We maintain the same non zero parameters and cluster index while quantizing, thus leaving the networks pruned and clustered. In particular, only the non zero values and the codebook values are quantized.

In order to perform our experiments, we first train the networks to achieve a similar accuracy performance as the one reported in literature. We then proceed to prune them to have more than 87% of

[1]Han *et al.* in [29] define clustering, *i.e.* weight-sharing, as "quantization". In this work we instead use the term *quantization* for indicating the approximation of the network parameters to fixed point representation.

their values set to zero while maintaining an accuracy drop below 2% after finetuning, *i.e.* retraining. After pruning we cluster the weight values in codebooks of at most 128 centroids, still maintaining a 2% accuracy drop after finetuning.

In order to test the sensitivity of the networks, Caffe Macchiato is able to inject single and multi bit faults in any network layer parameters which has not been zeroed, *i.e.* either weights or biases.

B. Application Data Regions

A deep neural network is typically composed of several layers that transform an input into either probabilities (classification problem) or real values (regression problem). A layer can include a set of weights and bias parameters that are used to process the layer input activations. During execution, these parameters will need to be available in main memory and so they can be susceptible to faults or attacks. In this chapter we limit our experiments to faults happening only offline in the network parameters, *i.e.* weights and bias values, and not at run time in the activations. The top part of Figure 5 provides a simplified view of how weight matrices are compressed and where the faults can happen.

We assume that a fault does not cause the application to crash, but rather that can possibly change the network output, *i.e.* that can affect its accuracy, thus causing Silent Data Corruption (SDC) faults.

For the baseline networks, we marked the layers' weights and bias parameters as a data region for faults to happen. For pruned networks, we further limit the data region to the non-zero parameters only, since the zero values are not required for the computations of a layer. If a network is also clustered, the non zero weights are substituted by a codebook and a list of indexes to the elements of the codebook (referred as *weight indexes*). Hence, for clustered networks, the data region consists of the codebook values, the weight indexes and the non zero bias parameters. We assume the codebook values being in a protected fault-free region of memory and so we are not testing faults happening in the codebook.

For non quantized networks, the weight, bias and codebook parameters are stored as single precision floating point values, whereas the weight indexes (in case of clustered networks) are assumed unsigned integers of bitwidth $log_2(k)$, where k is the number of clusters per layer. For quantized networks instead, we have the parameters quanzited to dynamic fixed point representation [30], [33].

C. Sensitivity Analysis and Methodology

In order to conduct a sensitivity analysis of DNNs, we follow the methodology described as follows. Given a network specification, we generate in Caffe Macchiato a series of network models which are compressed at different levels. In particular, out of a single network specification we obtain six different network configurations: a baseline network (B), a pruned network (P), a pruned and clustered network (P+C), a quantized network (Q), a quantized pruned network (Q+P) and a quantized pruned clustered network (Q+P+C). All compressed configurations exhibit a reduction in the accuracy of at most 3.1% with respect to the baseline.

Caffe Macchiato is then able to inject single and multi bit faults in any of the above network configurations by flipping bits at random locations in randomly selected parameters. Both the bit position and the targeted parameters for the faults to occur are uniformly distributed.

For the baseline configuration we use Caffe Macchiato for injecting a single-bit fault in a single parameter (either weight or bias) in any network layer. For simulating multi-bit flips instead, the framework swaps the value of two parameters, either two weight or two bias values. We then follows a similar approach for injecting faults in pruned networks, but only targeting a parameter chosen from the non zero ones. In case of pruned and clustered networks instead, we inject single- and multi-bit faults in either the *codebook indexes* or the bias parameters (both aren't zero). For single-bit flips in clustered networks, we flips a random bit in either a random weight index or a random non zero bias value. Multi-bit flips are instead performed by either swapping two weight indexes or by swapping two non zero bias values.

TABLE I: Description of fault types per network configuration. The faults can be injected in Baseline (B), Pruned (P) and Clustered (C) networks configurations. The location of faults can be either in weights (w) or in bias values (b). For pruned configurations, only the non-zero (NZ) values are selected.

Config	Loc	Single bit fault	Multi bit fault
(B)	(w)	bit flip in random weight	swap 2 random weights
	(b)	bit flip in random bias value	swap 2 random bias values
(P)	(w)	bit flip in random NZ weight	swap 2 random NZ weights
	(b)	bit flip in NZ random bias value	swap 2 random NZ biases
(P+C)	(w)	bit flip in random weight index	swap 2 random weight indexes
	(b)	bit flip in NZ random bias value	swap 2 random NZ biases
(Q)	(w)	bit flip in random quant weight	swap 2 random quant weights
	(b)	bit flip in random quant bias value	swap 2 random quant biases
(Q+P)	(w)	bit flip in random NZ quant weight	swap 2 random NZ quant weights
	(b)	bit flip in NZ random quant bias	swap 2 random NZ quant biases
(Q+P+C)	(w)	bit flip in random weight index	swap 2 random weight indexes
	(b)	bit flip in NZ random quant bias	swap 2 random NZ quant biases

A summary of the types of faults that can be simulated in Caffe Macchiato is reported in Table I. Since Quantization is an orthogonal technique, it can be applied to all the three reported configurations. In case of single bit flips, a fault can only happen within the bitwidth of the parameters.

V. Evaluation

In this section we present and evaluate the results of simulating faults in different networks, compressed in different configurations. For our analysis we chose a series of networks popular in the field of machine learning: a FCDN, LeNet-300-100 [20], and two CNNs: LeNet-5 [20] and CaffeNet [34]. Each network attempts to assign a class to the images from a test dataset. The more test images are correctly classified, the higher is the network accuracy.

The chosen networks have been pruned and clustered without a significant loss of accuracy, as reported in Table IIa. The configuration of the compressed networks after quantization is instead reported in Table IIb, which illustrates the bitwidth of the quantized parameters and the accuracy of the quantized networks. A summary of the compression ratios achieved for different configurations is showed in Table IIc. For our experiments, we report the accuracy drop as the averaged accuracy drop of 1000 fault injection tests. We report a negative drop in cases where a fault is actually improving the original non-faulty accuracy.

a) LeNet-300-100 on MNIST: We injected faults in LeNet-300-100, a network consisting of three fully connected layers classifying the MNIST database, which contains black and white images of hand-written digits. After injecting single-bit and multi-bit faults in both weights and bias in each layer of the baseline network, we do not see any particular drop in accuracy (the drops range from -0.03% to 0.05%). The pruned version of the network is also not affected by random single- and multi-bit faults, showing an accuracy drop between 0% and 0.07%. A similar scenario happens for the clustered network: the accuracy drop is insignificant when injecting single and multi bit flips in both weights and bias, oscillating between 0% and 0.05%. A detailed report of the accuracy drops for single precision parameters can be found in Appendix A, Figure 6.

Table III shows the result of injecting faults in the quantized network configurations. We can notice that a single bit-flip of the last fully connected layer can cause an average drop in accuracy of 3.95% if injected in weights and 2.68% if injected in bias. For the other layers and fault types we do not see any particular difference instead. A similar scenario happens with the quantized pruned network: a single bit flip fault in the last FC layer can cause a significant drop of 5.12% while a single bit flip in the `fc2` layer reduces the accuracy of 1.03% for injecting in weights and 1.50% in bias. Finally, regarding the clustered, pruned and quantized network, we experience significant drops in accuracy when injecting single bit flips faults in both weights and bias parameters, up to 3.13% and 2.72% for weight and bias respectively.

TABLE II: Different configurations for the analyzed networks and network accuracy scores. The reported configurations are: Baseline (B), Pruned (P), Clustered (C) and Quantized (Q).

(a) The percentage amount of non-zero elements (NZ) for the pruned configurations and the amount of clusters k for FC (k_{FC}) and CONV (k_{CONV}) layers. Notice that we cluster *after* pruning the networks and therefore the pruning percentage is the same in the two configurations.

Network	acc (B)	NZ	acc (P)	k_{CONV}	k_{FC}	acc (P+C)
LeNet-300-100	98.01%	9.7%	98.43%	-	8	97.17%
LeNet-5	99.13%	12.1%	99.09%	64	8|16	98.00%
CaffeNet	81.30%	11.7%	78.51%	128	8	79.11%

(b) Quantized networks accuracy and bitwidth for fully connected (BW_{FC}) and convolutional (BW_{CONV}) layers.

Network	BW_{CONV}	BW_{FC}	(Q)	(Q+P)	(Q+P+C)
LeNet-300-100	-	4	96.97%	96.60%	94.74%
LeNet-5	4	4	97.53%	97.20%	98.17%
CaffeNet	8	8	81.21%	78.20%	81.21%

(c) Original size and compression ratios for the selected networks in different configurations.

Network	Orig Size	(P)	(P+C)	(Q)	(Q+P)	(Q+P+C)
LeNet-300-100	8.14 MB	×10.3	×80.3	×8	×83.1	×86.3
LeNet-5	13.16 MB	×8.3	×80.8	×8	×66.4	×83.0
CaffeNet	2.73 MB	×8.5	×37.3	×4	×34.0	×41.6

TABLE III: Accuracy drop for LeNet-300-100 in configurations: Quantized (Q), Pruned (P) and Clustered (C). The accuracy of the faulty networks is averaged over 1000 tests. The faults are Single bit-flips (S) or Multi bit-flips (M), and are injected in Weights (w) or Bias (b) parameters.

Faulty Layer	Drops: (S in w)	(M in w)	(S in b)	(M in b)
(Q) fc1	0.11%	0.00%	0.26%	-0.06%
(Q) fc2	-0.10%	0.00%	-0.10%	-0.13%
(Q) fc3	3.95%	0.03%	2.68%	-0.02%
(Q+P) fc1	0.18%	0.00%	0.19%	0.00%
(Q+P) fc2	1.03%	0.00%	1.50%	0.06%
(Q+P) fc3	5.12%	0.00%	-	-
(Q+P+C) fc1	2.38%	0.00%	2.38%	0.00%
(Q+P+C) fc2	2.71%	0.00%	2.72%	0.02%
(Q+P+C) fc3	3.13%	0.01%	-	-

b) LeNet-5 on MNIST: LeNet-5 is a CNN composed of two CONV layers followed by two FC layers classifying the MNIST dataset, same as LeNet-300-100. We first injected faults in both the baseline, pruned and clustered networks with single precision floating point parameters. In presence of single- and multi-bit faults in either weights or bias parameters, we do not see any significant drop in accuracy. All drops remain below 1%, from as low as -0.06% up to 0.11%. More accurate accuracy drops results for single precision parameters can be found in Appendix A, Figure 8.

For the quantized configurations, the results of the fault injection tests are reported in Table IV. The quantized networks appear very resilient to multi bit flips, with almost all tests scoring no accuracy drops. We can instead notice significant drops in accuracy when injecting single bit flips, both in weights and bias parameters. For the quantized baseline, injecting faults in the first two CONV layers weights produces high drops of 1.55% and 3.92%, while injecting in the weights of the FC layers does not impact the accuracy. Single-bit faults happening in the bias parameters of any layer greatly affect the accuracy of the quantized baseline, resulting in average drops up to 7.38%.

When analyzing the quantized pruned network configuration, single-bit flips in the weight parameters largely influence the accuracy, causing average drops up to 17.84%. The pruned CaffeNet has most of

TABLE IV: Quantized LeNet-5. The accuracy of the faulty network is averaged over 1000 tests. The faults are Single bit-flips (S) or Multi bit-flips (M), and are injected in Weights (w) or Bias (b) parameters.

Faulty Layer	Drops: (S in w)	(M in w)	(S in b)	(M in b)
(Q) conv1	1.55%	0.00%	1.38%	0.00%
(Q) conv2	3.92%	0.00%	6.18%	0.00%
(Q) fc1	-0.81%	0.00%	7.38%	0.00%
(Q) fc2	-0.32%	0.00%	5.27%	0.00%
(Q+P) conv1	1.22%	0.00%	-	-
(Q+P) conv2	13.92%	0.00%	2.71%	-0.12%
(Q+P) fc1	13.87%	0.00%	-	-
(Q+P) fc2	17.84%	-0.01%	-	-
(Q+P+C) conv1	1.72%	0.00%	-	-
(Q+P+C) conv2	1.22%	0.00%	1.48%	0.13%
(Q+P+C) fc1	0.04%	0.00%	-	-
(Q+P+C) fc2	12.37%	0.01%	-	-

its bias parameters pruned and so the only layer where we injected faults in bias is `conv2`, causing a non-negligible drop of 2.71%. Lastly, injecting single bit flips in the codebook indexes of the quantized, pruned and clustered network leads to high accuracy drops, as high as 12.37% (except when injecting in the `fc1` layer).

c) CaffeNet on CIFAR10: We injected faults in different configurations of CaffeNet, a CNN made of three CONV layers followed by a final FC layer. CaffeNet is classifying the CIFAR10 dataset images, a more challenging database of colored images [35]. For the network configurations utilizing single precision floating point values, we could not see any significant accuracy drop after the fault injection tests. The drops are slightly higher compared to LeNet-300-100 and LeNet-5, but still well below 1% (between a minimum of -0.07% and a maximum of 0.53%). All the fault injection results for single precision parameters can be viewed in Appendix A, Figure 10.

We then proceeded to analyze the quantized configurations of CaffeNet. We followed the same approach as before and injected single- and multi-bit faults in the parameters of the layers, *i.e.* in weights and bias. The results of the tests are shown in Table V. We can see that multi-bit flips do not cause any particular accuracy drop, similarly to the previous cases (here we have an average drop between -0.01% and 0.55%). However, we can clearly notice a consistent drop in accuracy when injecting single bit flips, both in weights and bias parameters. In particular, for the quantized baseline network, the drop reaches up to 10.39% in weights and 3.65% in bias (the highest drops show up when targeting the last FC layer). We experience a similar trend for the quantized pruned configuration: the drop when injecting in weights is between 4.66% and 40.50%. It appears that random single-bit faults in the last FC layer can halve the original accuracy of the network. Please note that the pruning operation set to zero all the bias parameters and so no faults happening in bias were tested. Finally, the clustered, pruned and quantized CaffeNet shows high accuracy drops when injecting single bit faults in all layers but `conv3` (from 0.77% up to 13.56%). Multi-bit faults do not cause significant drops instead.

A. Discussion and Limitations

In this work we do not investigate the possible causes that led to the obtained results. In particular, when looking at networks utilizing single precision 32 bit floating point values, we suppose that the wider bitwidth of the values might help in mitigating single bit flips. In fact, the faulty bit position is uniformly distributed and the exponent field, where a bit flip can cause the largest magnitude change, is only 8 bit wide, as specified in the IEEE-754 standard.

Our best speculation regarding the fault injection runs that show low accuracy drops, is that the ReLU and Pooling layers might mask specific single bit-flip faults. In fact, the ReLU function, Equation 2, can zero out any negative value when $\gamma = 0$, which was the case for our experiments. Because of that, all single bit faults leading to a negative real value, even with very high magnitude, can be masked to zero.

TABLE V: Quantized CaffeNet-CIFAR10. The accuracy of the faulty network is averaged over 1000 tests. The faults are Single bit-flips (S) or Multi bit-flips (M), and are injected in Weights (w) or Bias (b) parameters.

Faulty Layer	Drops: (S in w)	(M in w)	(S in b)	(M in b)
(Q) conv1	0.36%	0.12%	0.42%	0.01%
(Q) conv2	7.31%	0.09%	0.78%	-0.01%
(Q) conv3	7.31%	0.07%	0.86%	0.00%
(Q) fc1	10.39%	0.05%	3.65%	-0.01%
(Q+P) conv1	4.66%	0.45%	-	-
(Q+P) conv2	14.91%	0.55%	-	-
(Q+P) conv3	13.15%	0.39%	-	-
(Q+P) fc1	40.50%	0.42%	-	-
(Q+P+C) conv1	4.70%	0.11%	-	-
(Q+P+C) conv2	13.56%	0.01%	-	-
(Q+P+C) conv3	0.77%	0.00%	-	-
(Q+P+C) fc1	3.42%	0.00%	-	-

Listing 1: Fault masking of `NaN` values through the `max` operation.

```
#define MAX(a, b) a > b ? a : b

MAX(1.0, NaN); // Evaluates to: 1.0 > NaN ? 1.0 : NaN, returns NaN
MAX(NaN, 1.0); // Evaluates to: NaN > 1.0 ? NaN : 1.0, returns 1.0
```

If the faulty value turns out to be a `NaN` (either because of a bit flip or as an accumulated result), both Max Pooling, Figure 3, and ReLU layers can mask it through their `max` operation. Listing 1 provides a simple example of the `max` implementation and the possible outcomes upon processing a `NaN` value.

Effectively, the IEEE-754 floating point standard does not specify the outcome of a comparison operation with a `NaN` [36]. However, our framework is developed in C++11, following the original Caffe implementation, which adheres to the IEC-559 standard [37], [38], which defines `false` as a return value for any comparison involving a `NaN`. Because of this, depending on "which side" the `NaN` value falls in the comparison, it can result in either a regular floating point number or a `NaN` value, thus eventually masking the fault.

Overall, the above conclusions seem to be supported by our findings. In fact, according to the results in Tables III, IV and V, the last FC layer in all of the tested networks, which is never followed by a Max Pooling layer nor a ReLU layer, generally appears to be, on average, the most sensitive to faults, *i.e.* leading to the highest accuracy drops.

VI. Conclusions

Our experiments suggest that using single precision floating point values ensures a high level of fault tolerance against random single- and multi-bit flips. Even if compressed, DNNs and in particular CNNs are able to correctly classify images in presence of faults and thus do not require any additional protection mechanism. Instead, significant drops in accuracy are happening to the quantized network configurations when injecting single bit flips. For the baseline quantized configuration, the networks are more tolerant to faults happening in their first layer, while are more affected if happening in the last layer. The drops of all pruned networks are higher than the ones of pruned and clustered networks, but both configurations show the highest drops when injecting in the last layer. We do not see a clear difference in drops caused when injecting single bit faults in weights or bias parameters, suggesting that both data regions are highly sensitive to faults. Overall, we observed that the faults causing the highest drops happen at the back of the network, *i.e.* in the last layers, whereas faults effects tend to be mitigated or compensated if happening in the first layer.

We can conclude that quantizing a DNN can significantly lower the fault tolerance of FCDNs and CNNs. In addition to this, our experiments show that further compressing the quantized networks by applying pruning and eventually clustering can lead to even higher losses in tolerance, thus making the networks requiring fault protection mechanisms. Based on our findings, as a future work we will be able to apply and test the Odd-ECC [2] protection mechanisms tailored to the identified more sensitive data regions.

REFERENCES

[1] M. Beyer, A. Morozov, K. Ding, S. Ding, and K. Janschek, "Quantification of the impact of random hardware faults on safety-critical ai applications: Cnn-based traffic sign recognition case study," in *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2019, pp. 118–119.

[2] A. Malek, E. Vasilakis, V. Papaefstathiou, P. Trancoso, and I. Sourdis, "Odd-ECC: on-demand DRAM error correcting codes," in *Proceedings of the International Symposium on Memory Systems*, 2017, pp. 96–111.

[3] G. Li, S. K. S. Hari, M. Sullivan, T. Tsai, K. Pattabiraman, J. Emer, and S. W. Keckler, "Understanding error propagation in deep learning neural network (dnn) accelerators and applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 1–12.

[4] F. F. d. Santos, P. F. Pimenta, C. Lunardi, L. Draghetti, L. Carro, D. Kaeli, and P. Rech, "Analyzing and Increasing the Reliability of Convolutional Neural Networks on GPUs," *IEEE Transactions on Reliability*, vol. 68, no. 2, pp. 663–677, 2019.

[5] J. J. Zhang, T. Gu, K. Basu, and S. Garg, "Analyzing and mitigating the impact of permanent faults on a systolic array based neural network accelerator," in *2018 IEEE 36th VLSI Test Symposium (VTS)*. IEEE, 2018, pp. 1–6.

[6] B. Reagen, U. Gupta, L. Pentecost, P. Whatmough, S. K. Lee, N. Mulholland, D. Brooks, and G.-Y. Wei, "Ares: A framework for quantifying the resilience of deep neural networks," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE, 2018, pp. 1–6.

[7] A. S. Rakin, Z. He, and D. Fan, "Bit-flip attack: Crushing neural network with progressive bit search," in *Proceedings of the IEEE International Conference on Computer Vision*, 2019, pp. 1211–1220.

[8] W. Choi, D. Shin, J. Park, and S. Ghosh, "Sensitivity based error resilient techniques for energy efficient deep neural network accelerators," in *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019, pp. 1–6.

[9] J. Ye, Y. Hu, and X. Li, "Hardware trojan in fpga cnn accelerator," in *2018 IEEE 27th Asian Test Symposium (ATS)*. IEEE, 2018, pp. 68–73.

[10] Y. Zhao, X. Hu, S. Li, J. Ye, L. Deng, Y. Ji, J. Xu, D. Wu, and Y. Xie, "Memory trojan attack on neural network accelerators," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019, pp. 1415–1420.

[11] J. Clements and Y. Lao, "Hardware trojan design on neural networks," in *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2019, pp. 1–5.

[12] Y. Liu, Y. Xie, and A. Srivastava, "Neural trojans," in *2017 IEEE International Conference on Computer Design (ICCD)*. IEEE, 2017, pp. 45–48.

[13] Y. Liu, L. Wei, B. Luo, and Q. Xu, "Fault injection attack on deep neural network," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2017, pp. 131–138.

[14] T. Gu, B. Dolan-Gavitt, and S. Garg, "Badnets: Identifying vulnerabilities in the machine learning model supply chain," *arXiv preprint arXiv:1708.06733*, 2017.

[15] T. Gu, K. Liu, B. Dolan-Gavitt, and S. Garg, "Badnets: Evaluating backdooring attacks on deep neural networks," *IEEE Access*, vol. 7, pp. 47 230–47 244, 2019.

[16] N. Khoshavi, C. Broyles, and Y. Bi, "Compression or corruption? a study on the effects of transient faults on bnn inference accelerators," in *2020 21st International Symposium on Quality Electronic Design (ISQED)*. IEEE, 2020, pp. 99–104.

[17] M. Qin, C. Sun, and D. Vucinic, "Robustness of neural networks against storage media errors," *arXiv preprint arXiv:1709.06173*, 2017.

[18] H. Guan, L. Ning, Z. Lin, X. Shen, H. Zhou, and S.-H. Lim, "In-place zero-space memory protection for cnn," *arXiv preprint arXiv:1910.14479*, 2019.

[19] B. E. Segee and M. J. Carter, "Fault tolerance of pruned multilayer networks," in *IJCNN-91-Seattle International Joint Conference on Neural Networks*, vol. 2. IEEE, 1991, pp. 447–452.

[20] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[21] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[22] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan, "Show and tell: A neural image caption generator," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 3156–3164.

[23] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," *arXiv preprint arXiv:1609.08144*, 2016.

[24] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *ICML*, 2010.

[25] D. Scherer, A. Müller, and S. Behnke, "Evaluation of pooling operations in convolutional architectures for object recognition," in *International conference on artificial neural networks*. Springer, 2010, pp. 92–101.

[26] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *International Conference on Learning Representations*, 2015.

[27] D. Blalock, J. J. G. Ortiz, J. Frankle, and J. Guttag, "What is the state of neural network pruning?" *arXiv preprint arXiv:2003.03033*, 2020.
[28] K. Guo, S. Zeng, J. Yu, Y. Wang, and H. Yang, "[DL] A survey of FPGA-based neural network inference accelerators," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 12, no. 1, pp. 1–26, 2019.
[29] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.
[30] P. Gysel, J. Pimentel, M. Motamedi, and S. Ghiasi, "Ristretto: A framework for empirical study of resource-efficient inference in convolutional neural networks," *IEEE transactions on neural networks and learning systems*, vol. 29, no. 11, pp. 5784–5789, 2018.
[31] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*, 2014, pp. 675–678.
[32] G. Kaur and M. M. Fuad, "An evaluation of protocol buffer," in *Proceedings of the ieee southeastcon 2010 (southeastcon)*. IEEE, 2010, pp. 459–462.
[33] D. Williamson, "Dynamically scaled fixed point arithmetic," in *[1991] IEEE Pacific Rim Conference on Communications, Computers and Signal Processing Conference Proceedings*. IEEE, 1991, pp. 315–318.
[34] "CaffeNet on CIFAR10," https://caffe.berkeleyvision.org/gathered/examples/cifar10.html, accessed: 2020-07-20.
[35] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," 2009.
[36] "Ieee standard for floating-point arithmetic," *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, 2019.
[37] "IEC 559:1989 Binary floating-point arithmetic for microprocessor systems," https://www.iso.org/standard/19706.html, accessed: 2021-02-19.
[38] "Working Draft, Standard for Programming Language C++," http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf, accessed: 2021-02-19.

APPENDIX

In this section we provide a detailed set of graphs reporting the distribution of the accuracy of the networks in presence of faults. For each experiment we collected 1000 accuracy samples, meaning that we injected 1000 faults per scenario. The results are grouped by layer and are divided according to the network configuration, being: Pruned (P), Clustered (C), Quantized to fixed point (Q) and their combinations. Faults are indicated either as Single or Multi bit flips in Weights (SW or MW) or Single of Multi bit flips in Bias (SB or MB).

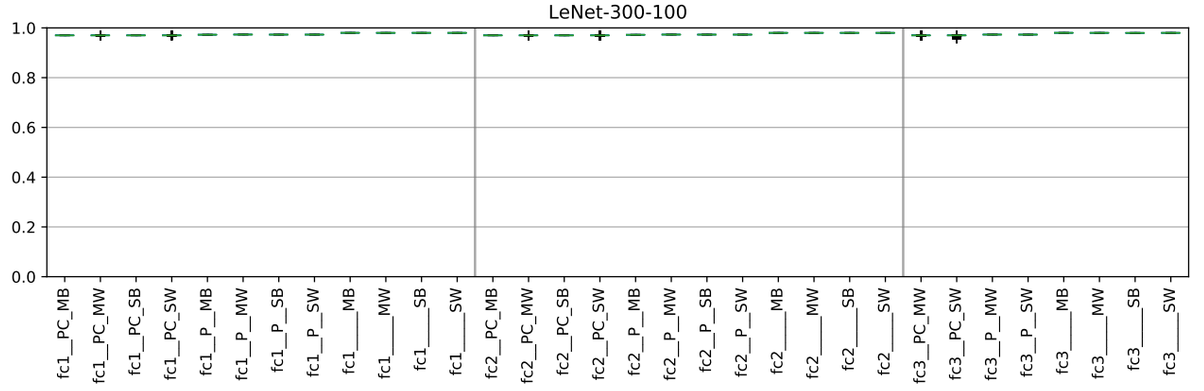

Fig. 6: Accuracy degradation distribution after injecting faults in LeNet-300-100.

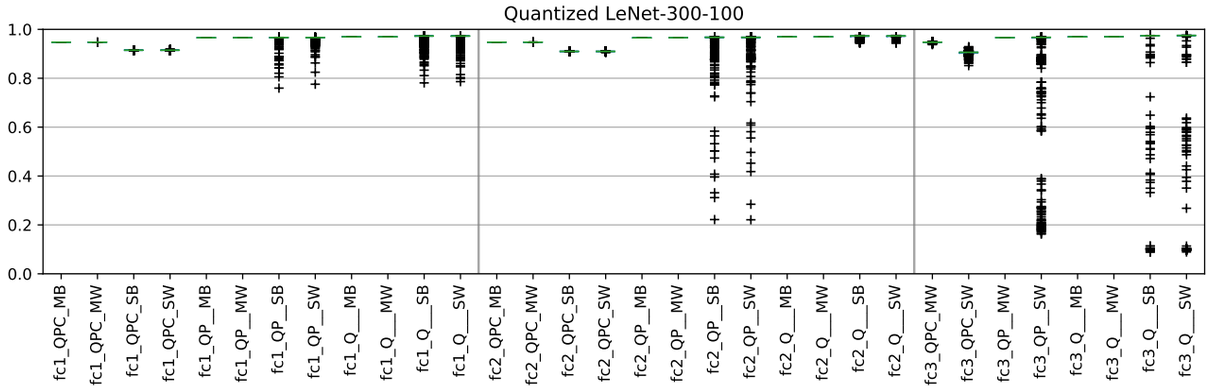

Fig. 7: Accuracy degradation distribution after injecting faults in quantized LeNet-300-100.

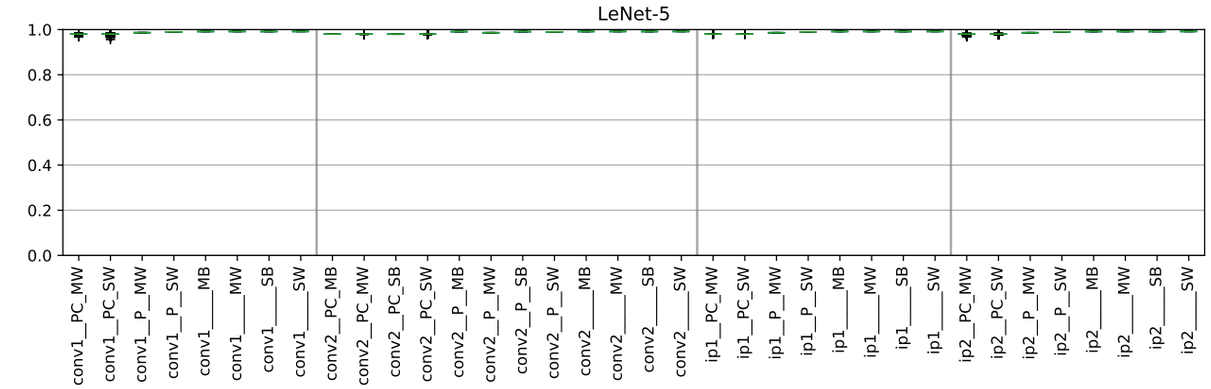

Fig. 8: Accuracy degradation distribution after injecting faults inLeNet-5.

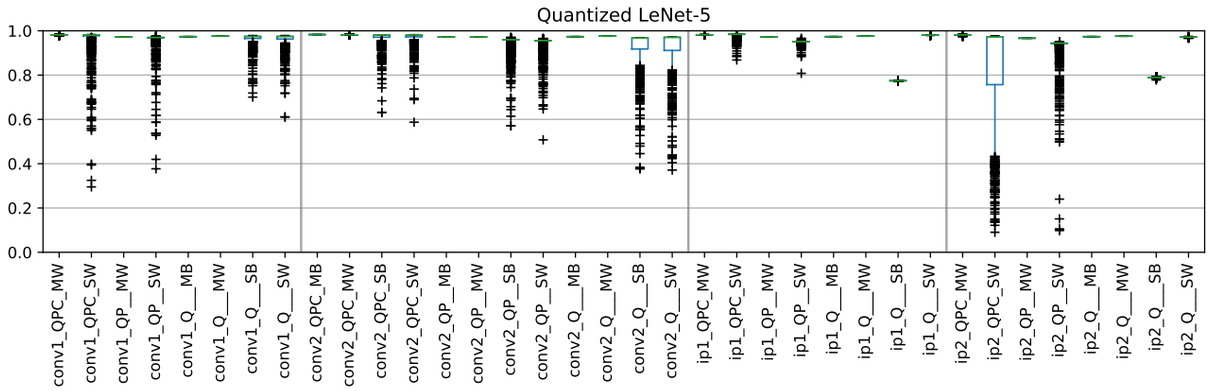

Fig. 9: Accuracy degradation distribution after injecting faults in quantized LeNet-5.

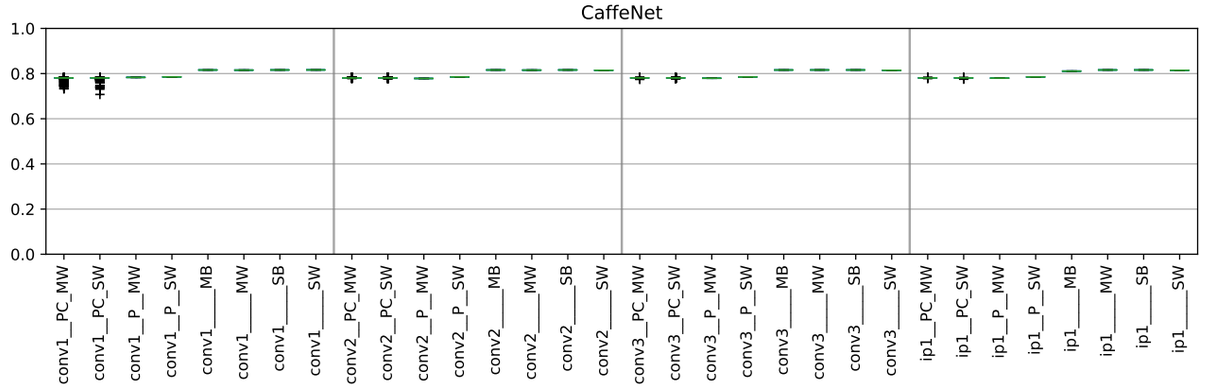

Fig. 10: Accuracy degradation distribution after injecting faults in CaffeNet.

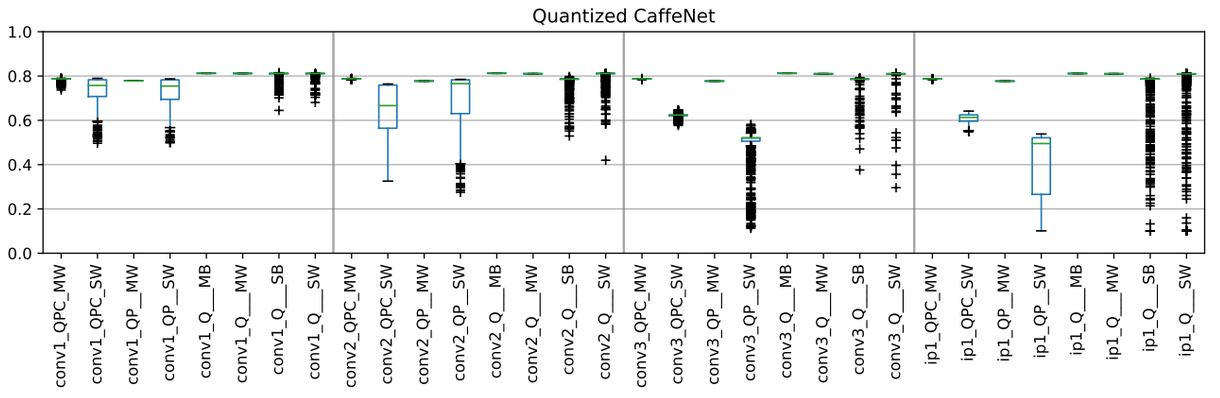

Fig. 11: Accuracy degradation distribution after injecting faults in quantized CaffeNet.